Efficient Computation of Mean Truncated Hitting Times on Very Large Graphs

Joel Lang and James Henderson
Computational Learning and Computational Linguistics Research Group
University of Geneva

5.12.2012

Abstract

Previous work has shown the effectiveness of random walk hitting times as a measure of dissimilarity in a variety of graph-based learning problems such as collaborative filtering, query suggestion or finding paraphrases. However, application of hitting times has been limited to small datasets because of computational restrictions. This paper develops a new approximation algorithm with which hitting times can be computed on very large, disk-resident graphs, making their application possible to problems which were previously out of reach. This will potentially benefit a range of large-scale problems.

1 Introduction

Efficient algorithms for graph-based learning have become central towards solving a range of large-scale learning problems. Prominent examples are ranking of web pages [Kleinberg, 1999, Page et al., 1999], collaborative filtering [e.g., Brand, 2005] and general-purpose methods for semi-supervised classification [e.g., Zhu et al., 2003]. Typically, graph vertices represent instances (e.g. webpages) and an edge indicates that two vertices are in some sense *close* or *similar*. Many of these problems involve graphs with hundreds of millions or billions of vertices and therefore call for methods which scale to such data quantities and in particular for methods which can be parallelized and executed on a cluster of machines.

Apart from the issue of scalability, a fundamental question underlying graph-based methods is how the information encoded in the edge weights of a graph can be exploited most effectively in order to transfer information between graph vertices, for example to propagate labels for classification tasks and similarly to propagate rank for ranking tasks. Interestingly, many successful methods can be related to the notion of graph random walks. In fact, *all* of the examples cited above either directly or indirectly correspond to computing a random walk on a graph. Moreover, previous work provides evidence that (dis-)similarity measures[1] based on random walk *hitting times* are particularly effective for many problems. For example, Mei et al. [2008] show that hitting times outperform alternative methods such as topic-sensitive pagerank [Haveliwala, 2003] for query suggestion, Fouss et al. [2007] sucessfully employ hitting times for collaborative filtering, Kok and Brockett [2010] generate paraphrases using hitting time based scoring, Yen et al. [2005] use a hitting time based measure for clustering and Gorelick et al. [2006] use hitting times for computing shape representations.

The mean hitting time of some vertex is the expected number of steps it takes for a random walk to reach that vertex, starting from some start vertex. One property which makes hitting times an appropriate dissimilarity measure in the aforementioned applications is their ability to reflect the overall connectivity structure of the graph, in contrast to measures such as the shortest path between two vertices. The hitting time will decrease when the number of paths from the start vertex to the target vertex increases, when the length of paths decreases or when the likelihood (weights) of paths increases. These properties are particularly important for problems where the graph edges encode a transitive (but not necessarily symmetric) relationship and must be assumed to be merely a sample of all plausible edges, possibly perturbed by noise.

[1] Here simply meaning a function which measures (dis-)similarity between vertices on some scale.

Consider for example the web graph, where a link from page a to page b will often indicate that b is relevant to a. This relationship is transitive (to some degree) and a page will link only to a sample of relevant pages. In such a setting, the hitting time for some vertex/page c and start vertex a will capture how relevant page c is for a and reflect all the evidence encoded in the graph about the relationship between the two.

So far, one drawback of methods based on hitting times compared to other methods based on random walks has been that computing the expected hitting times for a fixed start vertex to all other vertices is expensive, especially when using the standard iterative algorithm, as this requires computing a dense $n \times n$ matrix, where n is the number of vertices (details will follow below). For large graphs such as those mentioned above, computing this matrix is infeasible. Therefore, previous work has either investigated hitting times only on relatively small-scale problems with at most thousands of vertices [e.g., Fouss et al., 2007] or has been concerned with finding approximation algorithms which find upper and lower bounds and make use of sampling [Sarkar et al., 2008, Sarkar and Moore, 2007]. However, none of the proposed methods will scale to very large graphs which reside on disk rather than main memory.

This paper resolves the scalability problem by developing an approximation algorithm for computing mean truncated hitting times. The algorithm is space and runtime efficient, storing only 3 floating point numbers per vertex (in addition to the graph) and its runtime is linear in the number of edges. With our algorithm it becomes possible to compute hitting times for large-scale problems, where they could previously not be applied. Section 2 will provide some background and describe previous work. Then in Section 3 we describe our approximation algorithm. In Section 4 we will provide an empirical assessment of approximation accuracy.

2 Background and Previous Work

A random walk on a graph with vertices $V = \{1 \dots n\}$ is a discrete-time Markov chain $(X_t)_{t \in \mathbb{Z}^+}$, defined in terms of an initial distribution λ over V and a (row-)stochastic matrix P which captures the transition probabilities between vertices: $P_{ij} = p(X_{t+1} = j | X_t = i)$. For many applications, the transition matrix is a sparse matrix derived from the edge weights of the graph, which can be either directed or undirected.

The hitting time for some vertex $j \in V$ is the random variable $\Delta_{\lambda,j}$ over $\mathbb{Z}^+ \cup \{\infty\}$ which measures how long it takes until the random walk first hits j:

$$\Delta_{\lambda,j} = \inf\{t \geq 0 | X_t = j\}$$

where, as in Norris [1997], the infimum of the empty set is ∞. To simplify the discussion, we will initially assume that λ is a distribution with unit mass on vertex i, i.e., $\lambda = \delta_i$, and we will simply write $\Delta_{\delta_i,j} = \Delta_{ij}$. The vertex i will be called the *start vertex*.

The T-truncated hitting time, introduced by Sarkar and Moore [2007], is defined as

$$\Delta_{ij}^{(T)} = min(\Delta_{ij}, T)$$

Clearly, as T increases the truncated hitting time approaches the untruncated hitting time and asymptotically they are the same. For practical purposes, when hitting times are used for defining a (dis-)similarity measure it is usually sufficient or even superior [Sarkar and Moore, 2007] to compute truncated hitting times instead of untruncated hitting times, where typical values for T are between 10 and 20. The choice can be based on the mixing rate of the Markov chain, i.e., the rate at which the distribution over vertices converges towards the stationary distribution.

We are interested in computing the expected value of $\Delta_{ij}^{(T)}$, i.e., the mean truncated hitting time:

$$h_{ij}^{(T)} = E[\Delta_{ij}^{(T)}] = \sum_{t=0}^{T} tP[\Delta_{ij}^{(T)} = t] \tag{1}$$

The approach taken in Sarkar and Moore [2007] is based on the following recursive definition of hitting times, which is equivalent with the definition above:

$$h_{ij}^{(T)} = \begin{cases} 0 & i = j \vee T = 0 \\ 1 + \sum_{k \in V} p_{ik} h_{kj}^{(T-1)} \end{cases} \tag{2}$$

The problem with this approach is that, while computing $h_{ij}^{(T)}$ for all i and fixed j is relatively straightforward, computing $h_{ij}^{(T)}$ for all j and fixed start vertex i is computationally expensive and requires computing the full $n \times n$ matrix of mean truncated hitting times at intermediate steps of the computation. This can be seen from Equation 2, where in the second case $h_{kj}^{(T-1)}$ is required for all neighbors k in order to compute $h_{ij}^{(T)}$, and in order to compute $h_{kj}^{(T-1)}$ we need the $T-2$ truncated hitting times for all of k's neigbors, and so on. For large n computing a dense $n \times n$ matrix is intractable, which is why Sarkar and Moore [2007] have proposed a pruning scheme with which hitting times can be computed approximately.

Specifically, they derive upper and lower bounds which are precomputed and stored for each pair of close neighbors. The set of close neighbors is determined by iterative expansion (see Sarkar and Moore [2007] for details). Once the bounds for all close neighbors have been precomputed they can then be queried in order to compute bounds on the hitting time for an arbitrary pair of vertices. We will briefly review here how these bounds are computed.

Let $\mathcal{N}(i)$ denote the set of direct neighbors of i (reachable within one step), let $\mathcal{C}(j)$ denote a given set of vertices with short paths leading to vertex j and let $\mathcal{B}(j) \subset \mathcal{C}(j)$ denote a set of boundary vertices, which also have paths leading to vertices outside $\mathcal{C}(j)$. For a start vertex inside the close neighbors $i \in \mathcal{C}(j)$ the upper bound $\overline{h}_{ij}^{(T)}$ is computed as

$$\begin{aligned}\overline{h}_{ij}^{(T)} = 1 &+ \sum_{k \in \mathcal{C}(j) \cap \mathcal{N}(i)} P_{ik} \overline{h}_{kj}^{(T-1)} \\ &+ (1 - \sum_{k \in \mathcal{C}(j) \cap \mathcal{N}(i)} P_{ik})(T-1)\end{aligned}$$

and the lower bound $\underline{h}_{ij}^{(T)}$ is

$$\begin{aligned}\underline{h}_{ij}^{(T)} = 1 &+ \sum_{k \in \mathcal{C}(j) \cap \mathcal{N}(i)} P_{ik} \underline{h}_{kj}^{(T-1)} \\ &+ (1 - \sum_{k \in \mathcal{C}(j) \cap \mathcal{N}(i)} P_{ik})(1 + \min_{l \in \mathcal{B}(j)} \underline{h}_{lj}^{(T-2)})\end{aligned}$$

Once these bounds have been computed for each pair of close neighbors, the upper bound for a start vertex outside the close neighbors $i \notin \mathcal{C}(j)$ is simply T and the lower bound is

$$\underline{h}_{ij}^{(T)} = 1 + \min_{k \in \mathcal{C}(j)} \underline{h}_{kj}^{(T-1)}$$

While their algorithm can help reduce the storage requirements, storing all pairs of close neighbors and precomputing bounds for them is likely to be intractable for very large graphs and in the worst case requires $O(n^2)$ space and time. To improve performance Sarkar et al. [2008] resort to sampling for computing the hitting times from a start vertex, which is combined with the pruning scheme into an algorithm for computing the approximate top-k nearest commute time neighbors for a given query vertex. However, a sampling-based approach becomes inefficient when the graph does not fit into main memory, because repeated random access to disk will result in thrashing.

In the following section we will present an algorithm for approximately computing $h_{ij}^{(T)}$ for all j and fixed start vertex i which runs in $O(T|E| + Tn)$ time, where $|E|$ is the number of graph edges, and which stores only $3n$ additional floating point numbers as opposed to the n^2 floating point numbers stored in the conventional approach. Moreover, the algorithm is straightforward to parallelize for example within the map-reduce paradigm. Therefore, our approach is well-suited for disk-resident graphs, in contrast to sampling-based approaches.

3 Efficient Computation of Hitting Times

Our approach is based on the direct definition of hitting times given by Equation 1, rather than the recursive definition in Equation 2. The t-step transition matrix is written as P^t and accordingly the t-step transition

probabilities are written as P_{ij}^t. Assuming that the start vertex i is fixed, we must compute $P[\Delta_{ij}^{(T)} = t]$ for each j and t, in order to evaluate the sum in Equation 1. For compactness we will write

$$P_{ij}^{*t} = P[\Delta_{ij}^{(T)} = t] \qquad (t < T).$$

For T we have

$$P_{ij}^{*T} = 1 - \sum_{t=0}^{T-1} P_{ij}^{*t}$$

For the non-trivial case where $i \neq j$ we can write

$$P_{ij}^{*t} = P_{ij}^t - \sum_{k=1}^{t-1} P_{ij}^{*k} P_{jj}^{t-k} \tag{3}$$

Proof: Let S be the set of all paths of length t starting at i. Let $S_j^k \subset S$ be the set of paths which pass j the first time after k steps and end in j. The set $S_j \subset S$ of *all* paths ending in j is then given by $S_j = \cup_{k=1}^t S_j^k$. Since the sets S_j^k are mutually disjoint we have

$$P[S_j] = P[\cup_{k=1}^t S_j^k] = \sum_{k=1}^{t} P[S_j^k]$$
$$P[S_j^t] = P[S_j] - \sum_{k=1}^{t-1} P[S_j^k]$$

Now substituting $P[S_j] = P_{ij}^t$, $P[S_j^t] = P_{ij}^{*t}$ and $P[S_j^k] = P_{ij}^{*k} P_{jj}^{t-k}$ for $k < t$ we obtain Equation 3.

Consider again Equation 3: computing the values P_{ij}^t for all j and fixed i can be done in a space-efficient manner, by iteratively multiplying the transition matrix with the state-distribution vector at $t-1$. In contrast, computing the values P_{jj}^t requires us to compute all of the diagonal entries of P^t, which if done exactly requires computing the full t-step transition matrix for each $t < T$. Again, for large n this intractable.

We could at this point resort to sampling in order to estimate the P_{jj}^t with any desired accuracy with high probability. Let X_i^t be the Bernoulli random variable which indicates whether a random walk starting at i hits i after t steps, with probability $P[X_i^t = 1] = E[X_i^t] = p_{ii}$. We can obtain independent samples $\{x_{i0}^t \ldots x_{iL}^t\}$ of this random variable by sampling L (independent) random walks, each of length T since we are interested in all $t < T$. We will write $\mu_i^t = \frac{1}{L}\sum_{l=1}^{L} x_{il}^t$ for the empirical mean obtained from the sample. Then, using Hoeffding bounds we have

$$P[|\mu_i^t - p_{ii}^t| \geq \varepsilon] \leq 2\exp\left(-2\varepsilon^2 L\right)$$

Thus in order to have an ε-correct estimate of p_{ii}^t with at least probability $1-\rho$ we must sample at least $L = \frac{1}{2\varepsilon^2}\log\left(\frac{2}{\rho}\right)$ random walks for each vertex. Note that approximation errors to p_{ii}^t will be reduced when computing P_{ij}^{*t} according to Equation 3 since they are multiplied by $P_{ij}^{*k} \leq 1$ and errors will tend to be cancelled out by the sum. On the other hand, they may be amplified by factor t when computing the actual expectation according to Equation 1.

As was mentioned above, a major disadvantage of a sampling-based approach is that it is inefficient when the graph does not fit into main memory, because repeated random access to disk will result in thrashing. In the following we will therefore avoid sampling and discuss an alternative algorithm with which approximate hitting times can be computed very efficiently. In practice there are many situations where obtaining a loose estimate of hitting times is sufficient. Our algorithm essentially renders sampling the P_{jj}^t unnecessary and therefore avoids the most expensive part of the computation of the sampling approach.

3.1 The Approximation

The approximation is based on the following equation

$$P_{ij}^{*t} = P[(X_t = j) \wedge (X_{t-1} \neq j) \ldots \wedge (X_1 \neq j) | X_0 = i] \tag{4}$$

Algorithm 1: Iterative, space-efficient computation of approximate mean truncated hitting times

input : vertices V
transition matrix P (sparse)
start vertex i
output: $h_j^{(T)} \approx h_{ij}^{(T)}$ for each $j \in V$

1 $h^{(0)} \leftarrow \mathbf{0}$
2 $p^{(0)} \leftarrow \delta_i$
3 $f^{(0)} \leftarrow \mathbf{1} - p^{(0)}$
4 **for** $t = 1 \ldots T-1$ **do**
5 $\quad p^{(t)} \leftarrow P^\top p^{(t-1)}$
6 $\quad h^{(t)} \leftarrow h^{(t-1)} + t(p^{(t)} \circ f^{(t-1)})$
7 $\quad f^{(t)} \leftarrow f^{(t-1)} \circ (\mathbf{1} - p^{(t)})$
8 **end**
9 $h^{(T)} \leftarrow h^{(T-1)} + T f^{(T-1)}$

We can then make the simplifying assumption that the events occurring in the conjunction are (almost) independent. The larger the graph, the larger t and the larger the mixing rate of the Markov chain, the more accurately this assumption holds. Intuitively, for many graphs, especially those representing 'real-world' application data, knowing that we are not in state j at some point of the Markov chain will provide only very little information on average about where we will be in the following states. This leads to the approximation:

$$P_{ij}^{*t} \approx P_{ij}^t \prod_{k=0}^{t-1} (1 - P_{ij}^k)$$

Likewise, for $t = T$ we can use

$$P_{ij}^{*T} \approx \prod_{k=0}^{T-1} (1 - P_{ij}^k)$$

Putting everything together results in the following approximation scheme

$$h_{ij}^{(T)} \approx \sum_{t=0}^{T-1} t \left[P_{ij}^t \prod_{k=0}^{t-1} (1 - P_{ij}^k) \right] + T \left[\prod_{k=0}^{T-1} (1 - P_{ij}^k) \right]$$

Based on this equation we can then compute $h_{ij}^{(T)}$ for each j and fixed i according to Algorithm 1. The algorithm proceeds by iteratively computing the terms occurring in the equation above. It maintains three vectors h, p and f. The vector $h^{(t)}$ stores the sum up to term t for each possible target vertex j, s.t. the final result $h^{(T)}$ corresponds to the vector of approximate mean T-truncated hitting times of a random walk starting at i. The vector $p^{(t)}$ stores the distribution over vertices of the random walk after t steps, i.e., it stores the $i-th$ row of the t-step transition matrix P^t. Finally, the vector $f^{(t)}$ stores the product $\prod_{k=0}^{t}(1 - P_{ij}^k)$ for each j. We have used the notation $a \circ b$ to denote the component-wise multiplication of two (same-length) vectors a and b.

The space required by our algorithm is simply the space for the three vectors h, p and f, giving a total of $3n$ floating point numbers (excluding storage of the graph, which resides on disk). The main computational load of the algorithm stems from the matrix-vector multiplication in Line 5 between the transition matrix P and the current distribution p. If P is sparse, we can resort to sparse matrix-vector multiplication methods, which requires computing as many multiplications as there are non-zero components in P. Typically, P has a non-zero component for each edge of the graph $G = (V, E)$, so each iteration requires $|E| + 3n$ multiplications and $|E| + 2n$ additions. The total runtime is therefore $O(T|E| + Tn)$. Importantly, devising a map-reduce version of the algorithm is straightforward, since matrix-vector multiplication can be implemented as a map-reduce operation [see Rajaraman and Ullman, 2010]. Finally, note that the algorithm can be easily applied in the case where, instead of a single start vertex, one is given a distribution λ over start vertices. All that needs to be changed is the initialization $p^{(0)} \leftarrow \lambda$.

	Sparse 1			Sparse 2			Dense		
	10	100	1000	10	100	1000	10	100	1000
avg err	0.0433	0.0003	0.0001	0.0423	0.0004	0.0001	0.0134	0.0002	0.0000
max err	0.2863	0.0122	0.0269	0.2621	0.0140	0.0227	0.0420	0.0005	0.0000
avg inv	0.0153	0.0049	0.0036	0.0163	0.0021	0.0016	0.0512	0.0110	0.0013
max inv	0.0422	0.0080	0.0041	0.0430	0.0041	0.0019	0.1156	0.0159	0.0015

Table 1: Approximation accuracy on small synthetic graphs.

3.2 Higher-Order Approximations

Our approximation was based on Equation 4, where we assumed (approximate) independence between events in the conjunction. While this results in a particularly efficient algorithm, we can improve the approximation accuracy by weakening this assumption and instead assuming d-th order Markovian dependencies between the events. For example for order d=1 we have

$$\begin{aligned} P_{ij}^{*t} =& P[(X_t = j) \wedge (X_{t-1} \neq j) \ldots \wedge (X_1 \neq j) | X_0 = i] \\ \approx& P[X_t = j | X_{t-1} \neq j, X_0 = i] \ldots P[X_1 \neq j | X_0 = i] \\ =& (P_{ij}^t - P_{ijj}^t) \prod_{k=2}^{t-1} \left(1 - \frac{P_{ij}^{k-1} - P_{ijj}^k}{1 - P_{ij}^k} \right) \end{aligned}$$

where $P_{ijj}^t = P(X_t = j, X_{t-1} = j | X_0 = i)$. Thus for order d=1 this improves the approximation accuracy without changing the asymptotic space and runtime complexity of our algorithm. However, for $d > 1$ it would in general require computing dense $n \times n$ matrices.

4 Approximation Accuracy

In this section we will demonstrate that our approximation algorithm empirically results in accurate estimates of hitting times and will usually induce a ranking very close to the one produced by exact hitting times. Here we will conduct experiments on small synthetic graphs on which hitting times are exactly computable.

Specifically we will use both sparse and dense directed graphs with 10, 100 and 1000 vertices respectively. For the sparse graphs we will generate 20, 1000 and 10000 edges respectively. Given the number of vertices and edges, the first type of sparse graph (SP1) is generated by first randomly sampling an incoming and outgoing edge for each vertex and then sampling additional edges uniformly at random until the number of total edges is reached (if a sampled edge is already present, the sampling step is repeated). The second type of sparse graph (SP2) is generated by again first sampling an incoming and outgoing edge for each vertex. Then further edges are added by first sampling the target vertex of an edge, whereby the probability of choosing a vertex is proportional to the number of incoming edges it already possesses and then sampling a source vertex uniformly at random. This tends to accumulate edges at certain vertices and results in a different type of sparse graph. Again, the process is repeated until the desired number of edges is reached. In both cases edges receive unit weights. For generating the dense graph (DEN) we create a fully connected, directed graph and sample weights from a uniform distribution.

We compute the following scores: the average and maximum relative error of the approximation computed over all vertex pairs. If h_{ij} is the exact value and $\hat{h}_{ij}$ is the approximation then the relative error is defined as $|\frac{h_{ij} - \hat{h}_{ij}}{h_{ij}}|$. We will also consider the rankings generated by the exact and approximate hitting times and compute the proportion of vertex pairs which are ranked differently in the two rankings, i.e., the relative number of inversions in the approximate ranking compared to the exact ranking. Small values indicate that the rankings are similar and a random ranking would result in around 50% inversions. We consider the exact and approximate rankings for each possible start vertex and again report the average and maximum

inversions. For each graph type we generate 30 random graphs and report the aggregate scores (average and maximum) over all of these graphs in Table 1.

4.0.1 Results

Results are shown in Table 1. As expected, the approximation accuracy is higher for dense graphs than for sparse graphs, because the mixing rate of the Markov chain is higher and therefore the independence assumption underlying our approximation is less violated. The results also confirm that the relative error decreases with the number of vertices and is already quite small even for small graphs. As in practice our algorithm will be applied to graphs several orders of magnitudes larger, we can expect very high approximation accuracies on these graphs. Moreover, the ranking induced by the approximate hitting times are nearly identical to those produced by exact hitting times, and thus in many applications will constitute a valid replacement.

References

M. Brand. A Random Walks Perspective on Maximizing Satisfaction and Profit. In *Proceedings of the SIAM International Conference on Data Mining*, 2005.

F. Fouss, A. Pirotte, J. Renders, and M. Saerens. Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, 2007.

L. Gorelick, M. Galun, E. Sharon, R. Basri, and A. Brandt. Shape Representation and Classification Using the Poisson Equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1991 –2005, 2006.

T. Haveliwala. Topic-sensitive PageRank: a Context-sensitive Ranking Algorithm for Web Search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.

J. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46(5):604–632, 1999.

S. Kok and C. Brockett. Hitting the Right Paraphrases in Good Time. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010.

Q. Mei, D. Zhou, and K. Church. Query Suggestion Using Hitting Time. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, 2008.

J. Norris. *Markov Chains*. Cambridge University Press, 1997.

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, InfoLab, Stanford University, 1999.

A. Rajaraman and J. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2010.

P. Sarkar and A. Moore. A Tractable Approach to Finding Closest Truncated-commute-time Neighbors in Large Graphs. In *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, 2007.

P. Sarkar, A. Moore, and A. Prakash. Fast Incremental Proximity Search in Large Graphs. In *Proceedings of the 25th International Conference on Machine Learning*, 2008.

L. Yen, D. Vanvyve, F. Wouters, F. Fouss, M. Verleysen, and M. Saerens. Clustering Using a Random Walk Based Distance Measure. In *Proceedings of the European Symposium on Artificial Neural Networks*, 2005.

X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions. In *Proceedings of the International Conference on Machine Learning*, 2003.